PGI® Compilers and Tools on the Cray XT5 Systems

11MAY 10

OLCF, NICS to Host Spring Hex-Core Workshop, User's Meeting

Dave Norton
dave.norton@pgroup.com
530.544.9075
www.hpfa.com

Craig Toepfer
craig.toepfer@pgroup.com
503.682.2806
www.pgroup.com

The Portland Group

Compiling codes with PGI

PGI is the default compiler on the XT5 systems.

Cray supplies wrappers to all of the compilers on the system so that the Fortran compiler is always invoked as “ftn”, the C compiler as “cc”, and C++ as “CC” regardless of the actual compile vendor being used.

```
> module list

  pgi/10.2.0

> ftn -V foo.f -o foo

  pgfortran 10.2-0 64-bit target on Linux -tp barcelona-64
```

Or you can call the compiler directly with “pgfortran” but you won’t get the Cray library wrappers for use in the XT5 system

Using a different version of PGI

On the Cray, to change the version of the PGI compiler, you need to switch modules:

```
> module switch pgi/10.2.0 pgi/9.0.4

> ftn -V foo.f -o foo

  pgf90 9.0-4 64-bit target on Linux -tp barcelona
```

On your workstation, if you have multiple versions of PGI installed, you can invoke a different version of the compiler through the compile driver:

```
> pgfortran -V9.0-4 hello.f -o hello

  pgfortran 9.0-4 64-bit target on Linux -tp istanbul-64
```

Changing target processors

The PGI compile driver by default compiles for the processor on which the compilation takes place. The driver allows you to easily cross compile for another target processor:

```
> pgfortran -V foo.f -o foo -tp istanbul-64

  pgfortran 10.4-0 64-bit target on Linux -tp istanbul-64
```

The Cray compile driver appears to prevent, or not support, this capability as it is set up to cross-compile by default. If you want to target an executable for istanbul-64, try creating *.o's using pgfortran directly, then use ftn to link.

```
> pgfortran -V -c foo.f -tp istanbul-64

> ftn -V foo.o -o foo
```

Basic levels of scalar optimization

```
> ftn  foo.f -o foo
```

Invoking the compiler with no flags for optimization will set the scalar optimization level to 1 if –g is not specified.

```
> ftn -g foo.f -o foo
```

Invoking the compiler with no flags for optimization will set the scalar optimization level to 0 if –g is specified.

```
> ftn -O foo.o -o foo
```

Invoking the compiler with the -O flag for optimization will set the scalar optimization level to 2 regardless of whether –g is also specified. Optimization levels O0 through O4 perform increasing aggressive *scalar* optimizations

Basic levels of vector optimization

```
> ftn -fast foo.f -o foo
```

Invoking the compiler with the –fast (or –fastsse) flag sets common optimizations which include:

-O2	
-Munroll=c:1	
-Mnoframe	(gives the compiler another register)
-Mlre	
-Mautoinline	
-Mvect=sse	<= this is the vectorizer
-Mscalarsse	
-Mcache_align	
-Mflushz	
-Mpre	

Basic levels of vector optimization

Vectorization is the key to getting the best performance out of floating point intense codes. Current processors are capable of operating on 128 bits at a time. This means they can do 2 – double precision operations or 4 – single precision operations at the same time – as long as those operations can all be described by a single instruction (i.e. a vector operation).

AVX – coming by the end of the year, increases this to 256 bit wide units

The vectorizer performs the following operations:

- Loop interchange and loop splitting
- Loop fusion
- Memory-hierarchy (cache tiling) optimizations
- Generation of SSE instructions and prefetch instructions
- Loop peeling to maximize vector alignment
- Alternate code generation

Common impediments to vector optimization

There are several common coding issues that may prevent vectorization. The programmer may have enough knowledge to provide additional information to the compiler to work around these issues.

In C and C++ the compiler may not have enough information about the pointers passed into a subroutine to be able to determine that those pointers don't overlap. (-Msafeptr option or pragma or restrict keyword)

Function calls can be inlined to allow vectorization (-Minline)

Constants may be of the wrong type (-Mfcon)

Loops may be too long or too short. In both cases, additional options to the vectorizer may be successful in generating vector code.

-Msafeptr Option and Pragma

–M[no]safeptr[=all | arg | auto | dummy | local | static | global]

all	All pointers are safe
arg	Argument pointers are safe
local	local pointers are safe
static	static local pointers are safe
global	global pointers are safe

#pragma [*scope*] [no]safeptr={arg | local | global | static | all},…

Where *scope* is *global*, *routine* or *loop*

The Portland Group

Which level of optimization to start?

If you are just starting with a new code, we suggest that you try a short run of the code with optimization level –O2.

If the answers look good, then try the same run with the –fast flag.

If the answers are the same as the first run, use –fast as the basis for further optimizations. If the answers differ, try turning of optimizations one at a time until you find the optimization that is causing the difference. You can then track down in your code where that difference occurs and determine if it can be fixed, or if the optimization needs to be left turned off.

Turning off optimizations

Optimization flags are processed on the command line in the order in which they occur. For example - to turn on all –fast optimizations except loop redundant elimination:

```
> ftn -fast -Mnolre foo.o -o foo
```

Most optimizations can be turned on with the syntax –M*optimization*

Most optimizations can be turned off with the syntax -Mno*optimization*

Optimizations and debugging

Optimizations and debugging don’t always go hand in hand, however...

```
> ftn -fast -gopt foo.f -o foo
```

-gopt inserts debugging information without disabling optimizations. It is often helpful for tracking down a code bug that only appears in optimized code, or a bug that occurs far enough into a code that running the code with no optimizations takes a painful amount of time.

Generating tracebacks

Linux uses the backtrace system call to create the stacktrace when a fault or error occurs. The only requirement is to link with the -Meh_frame option:

```
 > pgfortran -Meh_frame -o x x.f90
```

Then before running the program, the following environment variable is set as follows:

```
> export PGI_TERM=trace
```

Generating tracebacks

Here is a sample traceback from within the PGI runtime.
(An attempt to deallocate an allocatable array more than one time):

```
0: DEALLOCATE: memory at (nil) not allocated
 ./x(__hpf_abort+0x7d) [0x40bb8d]
 ./x(__hpf_dealloc+0xeb) [0x40b57b]
 ./x(MAIN_+0x217) [0x408177]
 ./x(main+0x40) [0x407f40]
 /lib64/libc.so.6(__libc_start_main+0xf4) [0x2b877285e154]
 ./x [0x407e69]
```

Here is a sample traceback from a SEGV in user code:

```
Error: segmentation violation, address not mapped to object
  rax 000000005f45908, rbx 0000000000000001, rcx 0000000000187f9
  rdx 0000000000187f9, rsp 00007fffcdaef9a0, rbp 00007fffcdaef9a0
  rsi 00007fffcdaef9c4, rdi 00002ab2dd77e020, r8  00000000ffffffff
  r9  0000000000000000, r10 0000000000000022, r11 0000000000000246
  r12 0000000000000001, r13 00007fffcdaefae0, r14 0000000000000000
  r15 0000000000000000
 /lib64/libpthread.so.0 [0x2ab2dd1ebc10]
 ./y(init_+0x1f) [0x4081bf]
 ./y(MAIN_+0x9b) [0x407ffb]
 ./y(main+0x40) [0x407f40]
 /lib64/libc.so.6(__libc_start_main+0xf4) [0x2ab2dd468154]
 ./y [0x407e69]
```

What does *this* flag do?

There are too many compiler flags to remember all of their options. You can get help in several places:

```
> man pgfortran
```

`> pgfortran -fast -help` – gives help on -fast

Full PDF manuals are online in (e.g)

/opt/pgi/10.3.0/linux86-64/2010/doc

Manuals are also available at:

http://www.pgroup.com/resources/docs.htm

What exactly is being optimized?

Optimization is as much a user exercise as it is a compiler exercise. To see what the compiler thinks of your code, compile using the –Minfo flag.

```
> pgfortran -fast -Minfo foo.f -o foo
```

Use the information generated by –Minfo to help identify coding issues and locate places where code can be improved so the compiler can do an optimal job on it.

```
> pgfortran -Minfo -help
```

Use –Minfo to see which loops vectorize

```
> ftn -fast -Mipa=fast -Minfo -S graphRoutines.f90

localmove:
   334, Loop unrolled 1 times (completely unrolled)
   343, Loop unrolled 2 times (completely unrolled)
   358, Generating vector sse code for inner loop
   364, Generating vector sse code for inner loop
           Generating vector sse code for inner loop
   392, Generating vector sse code for inner loop
   423, Generating vector sse code for inner loop
```

Use –Mneginfo to see why things don't vectorize

The Portland Group

Additional compiler optimizations

The –fast flag is the 90/90 solution for code optimization. That is, it achieves about 90% of the possible performance for about 90% of the codes.

That means there are some additional areas that can be explored.

Interprocedural analysis can be helpful for C codes and Fortran codes without interface blocks. (Interface blocks are to the language specification what IPA is to the compiler)

```
> ftn -fast -Minfo -Mipa=fast foo.f -o foo
```

***If compiling and linking are done in separate steps, you must be sure to pass the IPA flag to the linker too.

IPA involves an additional pass of the compiler.

The Portland Group

Additional IPA optimizations

The suggested usage for IPA is to apply –Mipa=fast globally

The –Mipa flag has a *large* number of options that may be helpful in certain circumstances. These options are generally best applied to a specific subroutine to address a specific issue.

A couple of the more interesting flags include:

-Mipa=libopt   This allows recompiling and optimization of routines from libraries using IPA information. If you make extensive use of libraries in your code, try compiling those libraries with –Mipa=fast so that you have the option of using IPA when you link your application to that library

-Mipa=safeall This declares that all unknown procedures are safe.

Additional compiler optimizations

Several memory management options are available and may be beneficial depending on how your code accesses memory. *Smartalloc* tends to do a better job managing memory then standard Unix malloc.

Smartalloc can make use of “big pages”. Using big pages helps to minimize the number to TLB misses. This option tends to be helpful for codes that do a big initial allocate and then manage their own memory.

```
> ftn -fast -Minfo -Mipa=fast -Msmartalloc=huge foo.f -o foo
```

***-Msmartalloc must be used to compile main, and also to link the program

Additional compiler optimizations

Inlining can have a significant impact on application performance. It’s most dramatic effects tend to be on C++ codes which have many many small functions.

Inlining can be done at several different points in the compilation.

-Minline/autoinline	- during the regular compilation phase
-Mipa=inline	- during the recompile for IPA
Inline libraries	- created during the “make” process

The auto inliner is for C/C++ only. This enables inlining functios with the inline attribute. The suboptions control how the auto inliner operates.

-M[no]autoinline

Enable inlining of functions with the inline attribute.
-Mautoinline is implied with the -fast switch. The options are:

levels:n Inline up to n levels of function calls; the default is to inline up to 10 levels.

maxsize:n Only inline functions with a size of n or less. The size roughly corresponds to the number of statements in the function, though the correspondence is not direct. The default is to inline functions with a size of 100 or less.

totalsize:n

Stop inlining when this function reaches a size of n. The default is to stop inlining when a size of 8000 has been reached.

Creating and Using Inline Libraries

Use of -Minline/-Mextract to create an inline library. This works for all languages(C/C++/FORTRAN). To create an inline library with -Mextract do the following:

```
pgfortran -Mextract=lib:libfloat.il -c add.f90
pgfortran -Mextract=lib:libfloat.il -c sub.f90
pgfortran -Mextract=lib:libfloat.il -c mul.f90
pgfortran -Mextract=lib:libfloat.il -c div.f90
```

This creates an inline library name libfloat.il which can be used during compliation as follows:

```
pgf90 -fast -Minline=libfloat.il -c -Minfo -Mneginfo
       driver.f90
```

The -Minfo messages for this compile are:

```
test:
     14, Generated an alternate loop for the loop
         Generated vector sse code for the loop
     21, Generated an alternate loop for the loop
         Generated vector sse code for the loop
     22, add inlined, size=2, file add.f90 (2)
     33, Generated an alternate loop for the loop
         Generated vector sse code for the loop
     34, sub inlined, size=2, file sub.f90 (2)
     45, Generated an alternate loop for the loop
         Generated vector sse code for the loop
     46, mul inlined, size=2, file mul.f90 (2)
     57, Generated an alternate loop for the loop
         Generated vector sse code for the loop
     58, div inlined, size=2, file div.f90 (2)
```

As a result of inlining the functions add, sub, mul, and div the compiler was then able to vectorize the loops that contained those calls.

Use of -Mipa=inline to inline functions/subroutines. This works for all languages(C/C++/FORTRAN). Create the library using the -Mipa=inline option as follows:

```
pgfortran -Mipa=fast,inline -Minfo -Mneginfo -c add.f90
pgfortran -Mipa=fast,inline -Minfo -Mneginfo -c sub.f90
pgfortran -Mipa=fast,inline -Minfo -Mneginfo -c mul.f90
pgfortran -Mipa=fast,inline -Minfo -Mneginfo -c div.f90

ar cr libfloat.a add.o sub.o mul.o div.o
```

This creates a library named libfloat.a which can be used during compliation as follows(need to use the libinline suboption):

```
pgf90 -fast -Mipa=fast,inline,libinline -c -Minfo -Mneginfo
       driver.f90
pgf90 -fast -Mipa=fast,inline,libinline -o d driver.o
       libfloat.a
```

The -Minfo messages for this compile are:

```
test:
     14, Generated an alternate loop for the loop
         Generated vector sse code for the loop
     21, Loop not vectorized/parallelized: contains call
     33, Loop not vectorized/parallelized: contains call
     45, Loop not vectorized/parallelized: contains call
     57, Loop not vectorized/parallelized: contains call
IPA: Recompiling driver.o: stale object file
test:
      0, Pointer c is only set via allocate statements
         Pointer b is only set via allocate statements
         Pointer a is only set via allocate statements
         Function add does not write to any of its arguments
         Function add does not reallocate any of its arguments
         Function add does not reassociate any of its pointer arguments
         Function add does not reallocate any global variables
         Function add does not reassociate any global pointers
         Function add does not read any global (common/module) variables
         Function add does not write any global (common/module) variables
         Function sub does not write to any of its arguments
         Function sub does not reallocate any of its arguments
         Function sub does not reassociate any of its pointer arguments
         Function sub does not reallocate any global variables
         Function sub does not reassociate any global pointers
         Function sub does not read any global (common/module) variables
         Function sub does not write any global (common/module) variables
         Function mul does not write to any of its arguments
         Function mul does not reallocate any of its arguments
         Function mul does not reassociate any of its pointer arguments
         Function mul does not reallocate any global variables
```

Compiler optimizations and accuracy

There are a number of compiler options that offer the possibility of significant performance improvement at the expense of accuracy. If you are having numerical issues, you might tighten some restrictions.

`-Kieee` – floating point strictly conforms to IEEE 754 standard. (off by default)

`-Ktrap` – turns on the behavior of the processor when exceptions occur

`-Mdaz` – mode to treat IEEE denormalized input numbers as zero

`-Mflushz` – set SSE to flush-to-zero mode (on with –fast)

`-Mfprelaxed -` perform certain floating point operations using relaxed precision when it improves the speed. (This is the default mode on most other vendor's compilers)

Using more then one core

There are three general techniques for using more then one core for a computation. Of course, on large XT5 machines, all codes implement parallelism through MPI.

While most codes are MPI everywhere, some codes benefit by using the shared memory on the node through either automagic parallelizing by the compiler or/and OpenMP. OpenMP compilation is invoked with the –mp flag, automagic parallelization with the –Mconcur flag.

Environment variables which can effect OpenMP performance include:

OMP_SCHEDULE – can be static, dynamic, guided or auto

OMP_NUM_THREADS – specifies the number of threads to use

OMP_STACKSIZE – override the default stack size for new threads.

Profiling code

Cray provides some excellent tools for profiling using hardware counters.

PGI also provides some mechanisms for profiling of code. The simplest method is to use pgcollect. No special build process is needed, although compiling with –Minfo=ccff may provide useful feedback. This imbeds the –Minfo messages into the executable which can then be viewed with the performance profile.

Run your code as:

```
> pgcollect a.out
```

Then view the results with the GUI tool - pgprof

```
> pgprof -exe a.out
```

Profiling code

To get a general profile for an MPI code, you may wish to just profile one of the MPI processes. Running the code is where things change. Instead of launching the executable via mpiexec, launch a script instead:

```
 > mpiexec -np 2 ./doit
```

The doit script for code compiled and linked with MPICH2 might look like the following:

```
#!/bin/csh

if ($PMI_RANK == 0) then
   pgcollect ./test
else
  ./test
endif
```

After the run is complete, there will be only one pgprof.out file which can be viewed using:

```
>  pgprof -exe ./test pgprof.out
```

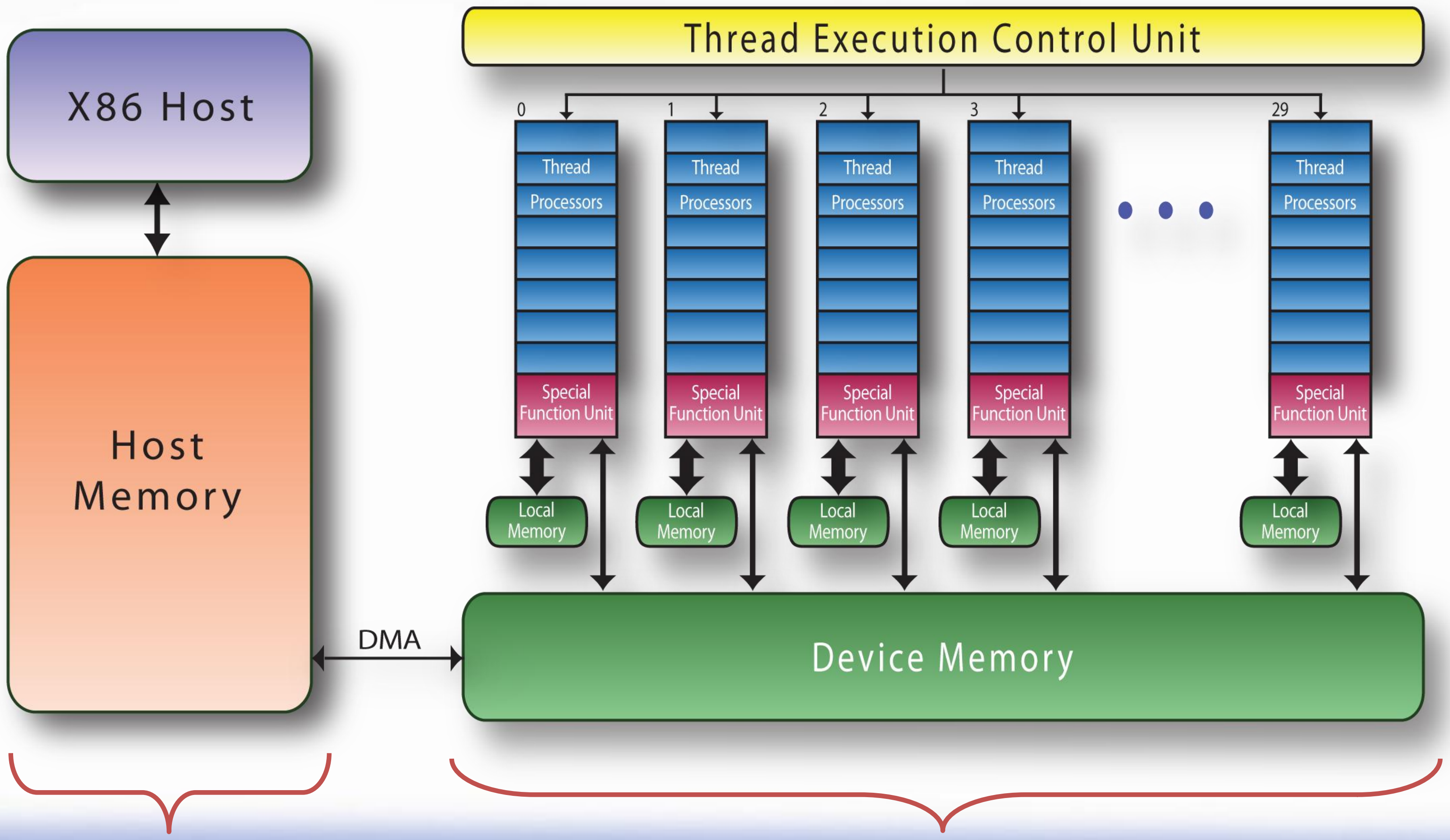
Extending Host-side x64 Compilers to Enable Incremental use of GPGPUs

The Portland Group

- **NVIDIA TESLA C1060**
  - Lots of available performance ~1 TFlops peak SP
  - Programming is a challenge
  - Getting high performance is lots of work
- **NVIDIA CUDA programming model and C for CUDA simplify GPGPU programming**
  - Much easier than OpenGL/DirectX, still challenging
  - PGI CUDA Fortran simplifies it even further
- **PGI Accelerator compilers do for GPU programming what OpenMP did for Posix Threads**

Emerging Cluster Node Architecture

Commodity Multicore x86 + Commodity Manycore GPUs



CPU Cores

GPU/Accelerator Cores

The Portland Group

Simple Fortran Matrix Multiply for an x64 Host

```
do j = 1, m
  do i = 1, n
    do k = 1,p
      a(i,j) = a(i,j) + b(i,k)*c(k,j)
    enddo
  enddo
enddo
```

Basic CUDA C Matrix Multiply Kernel for an NVIDIA GPU

```
extern "C" __global__ void
mmkernel( float* a,float* b,float* c,
          int la,int lb,int lc,int n,
          int m,int p )
{
    int i = blockIdx.x*64+threadIdx.x;
    int j = blockIdx.y;

    float sum = 0.0;
    for( int k = 0; k < p; ++k )
      sum += b[i+lb*k] * c[k+lc*j];
    a[i+la*j] = sum;
}
```

Optimized CUDA C Matrix Multiply Kernel

```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c, int la, int lb, int lc, int n, int m, int p )
{
    int tx = threadIdx.x;
    int i = blockIdx.x*128 + tx;  int j = blockIdx.y*4;
    __shared__ float cb0[128], cb1[128], cb2[128], cb3[128];

    float sum0 = 0.0, sum1 = 0.0, sum2 = 0.0, sum3 = 0.0;
    for( int ks = 0; ks < p; ks += 128 ){
      cb0[tx] = c[ks+tx+lc*j];      cb1[tx] = c[ks+tx+lc*(j+1)];
      cb2[tx] = c[ks+tx+lc*(j+2)]; cb3[tx] = c[ks+tx+lc*(j+3)];
      __syncthreads();
      for( int k = 0; k < 128; k+=4 ){
        float rb = b[i+lb*(k+ks)];
         sum0 += rb * cb0[k];    sum1 += rb * cb1[k];
          sum2 += rb * cb2[k];    sum3 += rb * cb3[k];
        rb = b[i+lb*(k+ks+1)];
         sum0 += rb * cb0[k+1]; sum1 += rb * cb1[k+1];
          sum2 += rb * cb2[k+1]; sum3 += rb * cb3[k+1];
        rb = b[i+lb*(k+ks+2)];
         sum0 += rb * cb0[k+2]; sum1 += rb * cb1[k+2];
          sum2 += rb * cb2[k+2]; sum3 += rb * cb3[k+2];
        rb = b[i+lb*(k+ks+3)];
         sum0 += rb * cb0[k+3]; sum1 += rb * cb1[k+3];
          sum2 += rb * cb2[k+3]; sum3 += rb * cb3[k+3];
      }
      __syncthreads();
    }
    a[i+la*j] = sum0;      a[i+la*(j+1)] = sum1;
     a[i+la*(j+2)] = sum2; a[i+la*(j+3)] = sum3;
}
```

Host-side CUDA C Matrix Multiply GPU Control Code

```
cudaMalloc( &bp, memsize );
cudaMalloc( &ap, memsize );
cudaMalloc( &cp, memsize );

cudaMemcpy( bp, b, memsize, cudaMemcpyHostToDevice );
cudaMemcpy( cp, c, memsize, cudaMemcpyHostToDevice );
cudaMemcpy( ap, a, memsize, cudaMemcpyHostToDevice );

dim3 threads( 128 );
dim3 blocks( matsize/128, matsize/4 );
mmkernel<<<blocks,threads>>>(ap,bp,cp,nsize,nsize,
                nsize,matsize,matsize,matsize);

cudaMemcpy( a, ap, memsize, cudaMemcpyDeviceToHost );

cudaFree( ap );
cudaFree( bp );
cudaFree( cp );
```

What is CUDA Fortran?

- CUDA Fortran is an analog to NVIDIA's C for CUDA
- CUDA Fortran was co-defined by PGI and NVIDIA and implemented in the PGI 2010 Fortran 95/03 compiler
- Includes support for the full CUDA programming model API and introduces intuitive Fortran language extensions to simplify host vs GPU data management
- Is supported on Linux, MacOS and Windows, including support within PGI Visual Fortran on Windows

CUDA Fortran
Matrix Multiply Host Routine

```
   . . .
   subroutine mmul( A, B, C )                         ! Host routine to drive mmul_kernel
      real, dimension(:,:) :: A, B, C
                                                      ! Declare allocatable device arrays
      real, device, allocatable, dimension(:,:) :: Adev,Bdev,Cdev
      type(dim3) :: dimGrid, dimBlock                 ! Define thread grid, block shapes
! Begin execution
      N = size( A, 1 )
      M = size( A, 2 )
      L = size( B, 2 )
      allocate (Adev(N,M), Bdev(M,L), Cdev(N,L))      ! Allocate device arrays in GPU memory
      Adev = A(1:N,1:M)                               ! Copy input A to GPU device memory
      Bdev = B(1:M,1:L)                               ! Copy input B to GPU device memory
      dimGrid = dim3( N/16, M/16, 1 )                 ! Define thread grid dimensions
      dimBlock = dim3( 16, 16, 1 )                    ! Define thread block dimensions
                                                      ! Launch mmul_kernel on GPU
      call mmul_kernel<<<dimGrid,dimBlock>>>( Adev, Bdev, Cdev, N, M, L)

      C(1:N,1:L) = Cdev                               ! Copy result C back to host memory
      deallocate( Adev, Bdev, Cdev )                  ! Free device arrays
   end subroutine mmul
end module mmul_mod
```

CUDA Fortran
Matrix Multiply GPU Kernel

```
    module mmul_mod                                       ! Module containing matrix multiply
      use cudafor                                         !   CUDA Fortran GPU kernel
    contains
      attributes(global) subroutine mmul_kernel( A, B, C, N, M, L )
      real :: A(N,M), B(M,L), C(N,L)
      integer, value :: N, M, L
      integer :: i, j, kb, k, tx, ty
      real, shared :: Asub(16,16), Bsub(16,16)            ! Declare shared memory submatrix temps
      real :: Cij                                         ! Declare C(i,j) temp for accumulations
 ! Begin execution
      tx = threadidx%x                                    ! Get my thread indices
      ty = threadidx%y                                    !
      i = blockidx%x * 16 + tx                            ! This thread computes
      j = blockidx%y * 16 + ty                            !   C(i,j) = sum(A(i,:) * B(:,j))
      Cij = 0.0
      do kb = 1, M, 16
         Asub(tx,ty) = A(i,ks+tx-1)                       ! Each of 16x16 threads loads one
         Bsub(tx,ty) = B(ks+ty-1,j)                       !   one element of ASUB & BSUB into
         call syncthreads()                               !   shared memory
         do k = 1,16                                      ! Each thread accumulates length 16
            Cij = Cij + Asub(tx,k) * Bsub(k,ty)           !   partial dot product into its Cij
         enddo
         call syncthreads()
      enddo
      C(i,j) = Cij                                        ! Each thread stores its element
                                                          !   to the global C array
    end subroutine mmul_kernel                            ! End CUDA Fortran GPU kernel routine
    . . .
```

The Portland Group

CUDA C vs CUDA Fortran

CUDA C	CUDA Fortran
supports texture memory	NO texture memory (yet)
supports Runtime API	supports Runtime API
supports Driver API	NO Driver API
cudaMalloc, cudaFree	allocate, deallocate
cudaMemcpy	assignments
OpenGL interoperability	NO OpenGL interoperability
Direct3D interoperability	NO Direct3D interoperability
arrays zero-based	arrays one-based
threadidx/blockidx 0-based	threadidx/blockidx 1-based
unbound pointers	allocatable are device/host
pinned allocate routines	pinned attribute

The Portland Group

PGI Accelerator Directive-based Fortran Matrix Multiply for x64+GPU

```
!$acc region
    do j = 1, m
      do k = 1, p
        do i = 1,n
          a(i,j) = a(i,j) + b(i,k)*c(k,j)
        enddo
      enddo
    enddo
!$acc end region
```

PGI Accelerator Compilers

```
void saxpy (float a,
float *restrict x,
float *restrict y, int n){
#pragma acc region
{
    for (int i=1; i<n; i++)
        x[i] = a*x[i] + y[i];
}
}
```

compile

Host x86 Code

```
saxpy:
    …
    movl    (%rbx), %eax
    movl    %eax, -4(%rbp)
    call    __pg_cu_init
    . . .
    call    __pg_cu_alloc
    …
    call    __pg_cu_uploadp
    …
    call    __pg_cu_paramset
    …
    call    __pg_cu_launch

    …
    Call    __pg_cu_downloadp
    …
```

+

GPU/Accelerator Code

```
static __constant__ struct{
    int tc1;
    float* _y;
    float* _x;
    float _a;
    }a2;
extern "C" __global__ void
pgi_kernel_2() {
  int i1, i1s, ibx, itx;
  ibx = blockIdx.x;
  itx = threadIdx.x;
  for( i1s = ibx*256; i1s < a2.tc1; i1s += gridDim.x*256 ){
    i1 = itx + i1s;
    if( i1 < a2.tc1 ){
      a2._x[i1] = (a2._y[i1]+(a2._x[i1]*a2._a));
    }
  }
}
```

link

Unified HPC Application

execute

… with no change to existing makefiles, scripts, programming environment, etc

The Portland Group

PGI Accelerator
Program Execution Model

- **Host**
  - **executes most of the program**
  - **allocates accelerator memory**
  - **initiates data copy from host memory to accelerator**
  - **sends kernel code to accelerator**
  - **queues kernels for execution on accelerator**
  - **waits for kernel completion**
  - **initiates data copy from accelerator to host memory**
  - **deallocates accelerator memory**
- **Accelerator**
  - **executes kernels, one after another**
  - **concurrently, may transfer data between host and accelerator**

PGI Accelerator Compute Region

- **Compute region directive**
  - **Fortran syntax**

```
!$acc region [clause [,clause]…]
…
!$acc end region
```

  - **C syntax**

```
#pragma acc region [clause [,clause]…]
{
…
}
```

PGI Accelerator Region Clauses

Clause	Region Scope / Type
if (*cond*)	compute
copy (*list*)	compute, data
copyin (*list*)	compute, data
copyout (*list*)	compute, data
local (*list*)	compute, data
updatein (*list*)	compute, data, executable
updateout (*list*)	compute, data, executable

PGI Accelerator Loop Mapping Clauses

Clause	Scope
host [(*width*)]	loop
parallel [(*width*)]	loop
seq [(*width*)]	loop
vector [(*width*)]	loop
private (*list*)	loop
kernel	loop
independent *	loop
unroll (*width*)*	loop
cache (*list*)*	loop

*** Not supported until PGI 10.6 (June 2010)**

Compiler-to-User Feedback

% pgfortran -fast -ta=nvidia -Minfo mm.F90

```
mm1:
     11, Generating copyout(a(1:m,1:m))
         Generating copyin(c(1:m,1:m))
         Generating copyin(b(1:m,1:m))
         Generating compute capability 1.0 binary
         Generating compute capability 1.3 binary
     12, Loop is parallelizable
     13, Loop is parallelizable
         Accelerator kernel generated
         12, !$acc do parallel, vector(16)
         13, !$acc do parallel, vector(16)
             CC 1.0 : 6 registers; 24 shared, 80 constant, 0 local memory bytes; 100 occupancy
             CC 1.3 : 6 registers; 24 shared, 80 constant, 0 local memory bytes; 100 occupancy
     16, Loop carried reuse of 'a' prevents parallelization
     17, Loop is parallelizable
         Accelerator kernel generated
         12, !$acc do parallel, vector(16)
         16, !$acc do seq
             Cached references to size [16x16] block of 'b'
             Cached references to size [16x16] block of 'c'
         17, !$acc do parallel, vector(16)
             Using register for 'a'
             CC 1.0 : 17 registers; 2072 shared, 84 constant, 0 local memory bytes; 33 occupancy
             CC 1.3 : 17 registers; 2072 shared, 84 constant, 0 local memory bytes; 75 occupancyh
```

The Portland Group

Loop Schedules

```
Accelerator kernel generated
  26, #pragma acc for parallel, vector(16)
  27, #pragma acc for parallel, vector(16)
```

- Vector loops correspond to threadidx indices
- Parallel loops correspond to blockidx indices
- The loop nest above is mapped to CUDA schedule:

  `<<< dim3(ceil(N/16),ceil(M/16)),dim3(16,16) >>>`
- PGI Accelerator compiler strip-mines to protect against very long loop limits

Summary

- **State of GPU Programming Tools generally** – options, capabilities, robustness, ease-of-use all developing rapidly

- **PGI CUDA Fortran** – based on a proven GPGPU programming model – and it’s Fortran

- **PGI Accelerator programming model** – higher level, easily approachable, incremental (loop level, not routine-level), 100% portable source code, suitable for industry standardization, Fortran and C

The Portland Group

Future Directions

- NVIDIA Fermi / CUDA 3.0 support – upon release of the HW
- Remaining features of the v1.1 PGI Accelerator programming model standard coming in PGI 10.6, June 2010
- New features to be added to the model, and then the compilers
  - Asynchronous data transfers and kernel execution
  - Support for multiple Accelerator devices
- PGI Accelerator Fortran/C full interoperability with CUDA Fortran/C
- Debugging – lots of work to do here, just on the compiler side
- Native PTX code generation on NVIDIA targets
- Optimizations – Planner, GPU memory hierarchy, Inlining, IPA-based
- New PGI Accelerator targets – multi-core x64, ATI, Larrabee all are candidates

Reference Materials

- **Understanding the CUDA Data Parallel Threading Model**

  http://www.pgroup.com/lit/articles/insider/v2n1a5.htm

- **CUDA Fortran Programming Guide and Reference**

  http://www.pgroup.com/lit/whitepapers/pgicudaforug.pdf

- **PGI Fortran & C Accelerator Programming Model**

  http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.1.pdf